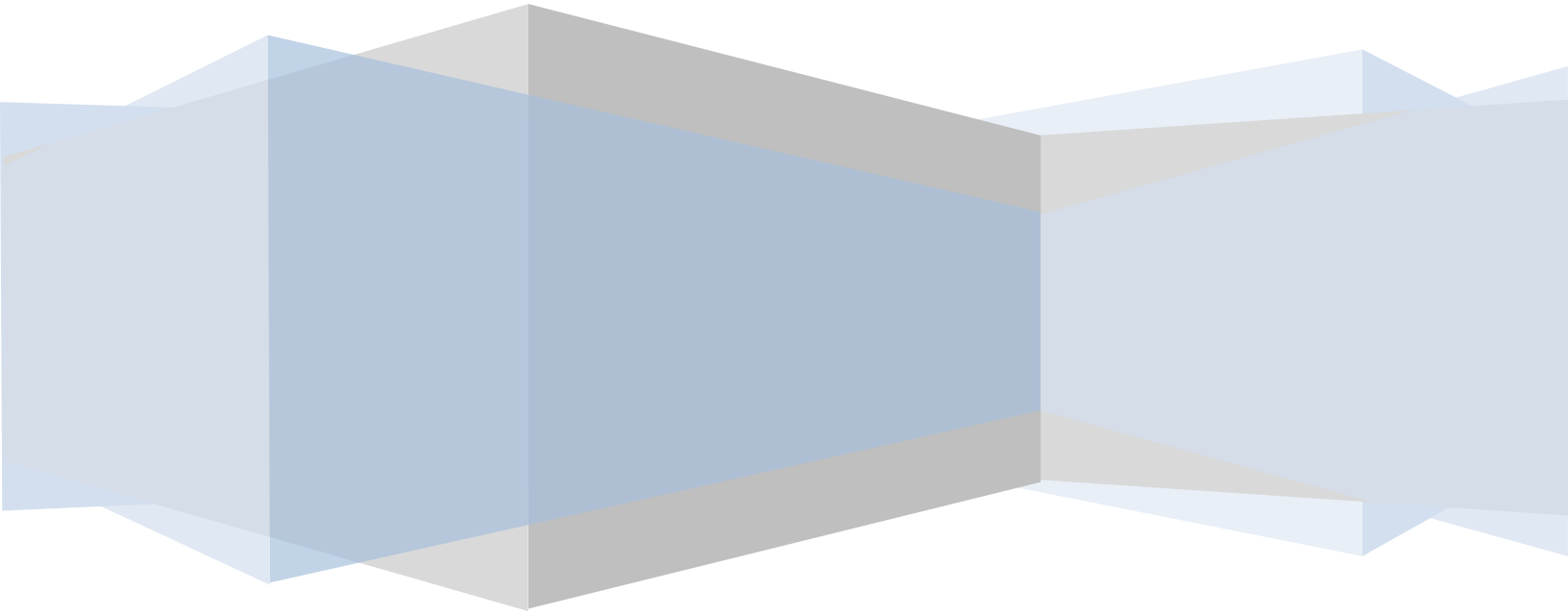
Readme for Adobe Director Hotfix 11.5.9.629



Contents

Pre-requisites for installing this version 3
Enhancements and bug fixes in this version 3
Flash Player 10.2 support 3
Parallax Mapping 3
Texture Transform [NEW] 4
Cube Mapping 5
Normal mapping 5
Normal Map Usage 6
Texture Transform [NEW] 7
Camera enhancements 7
Render to Texture 8
API Details 8
updateRTImageOnRender (Getter/Setter) 8
newTexture 9
renderToTexture 10
<RenderTextureObj>.image (Getter/Setter) 10
Bump mapping to normal mapping 11
bumpMapToNormalMap(<image>, bumpFactor) 11
Parameters 11
Additive shading 12
New/Modified APIs 12
Alpha sorting 12
<shader>.alphaSort 13
Enhancement of the visibility property 13
textureRepeat enhancements 14
Custom blend function for particle systems 14
Importing DAE models 15
Intel binary support 16

Pre-requisites for installing this version

This version (Hotfix#3 Build 629), should be applied only when one of the following software is installed on the machine:

- ✓ Director 11.5 GM Build 593
- ✓ Director 11.5 GM + Hotfix#1 Build 596
- ✓ Director 11.5 GM + Hotfix#2 Build 612

Enhancements and bug fixes in this version

The following enhancements are made in this Hotfix:

- ✓ **Parallax Mapping** [NEW]
    a. **Supports Texture Transform** [NEW]
- ✓ **Cube Mapping** [NEW]
- ✓ Normal Mapping
    a. **Supports Texture Transform** [NEW]
- ✓ Flash Player 10.2 support
- ✓ Importing Collada models
- ✓ Support for converting bump maps to normal maps
- ✓ Additive Transparency
- ✓ Support for a Mac projector as an Intel binary file
- ✓ Alpha Sorting

Flash Player 10.2 support

Adobe Shockwave Player now provides support for Adobe Flash Player 10.2.

Please refer to the Adobe Flash Player 10.2 documentation for more information.

Parallax Mapping

Information: This is a 3D enhancement.

Parallax mapping is an enhancement of Normal/Bump Mapping technique. This technique provides more apparent depth and greater realism with less impact on the performance. Parallax maps are exposed in Lingo as a new type of shader, which can be created by passing **#parallaxMap** as the type of shader to the **newShader** handler of the 3D cast member.

Usage

```
-- Creating the height map texture
heightMap = mem.newTexture("heightMap", #fromCastMember, member("heightMap_Image"))

-- Creating the diffuse texture
my_diffuse_texture = mem.newTexture("texture1", #fromCastMember,
member("texture_Image"))

-- Creating the specular texture
my_specular_texture = mem.newTexture("my_specular_texture", #fromCastMember,
member("specular_Image")

-- Creating a parallax map shader
sh = mem.newShader("myShader", #parallaxMap)

-- Assign the height map texture to the first layer
sh.textureList[1] = heightMap

-- Assign the diffuse texture to the second layer
sh.textureList[2] = my_diffuse_texture

-- Assign the specular texture to the third layer
sh.textureList[3] = my_specular_texture

-- Note: You can also use layers 4,5, and 6 for any other textures
-- Applying diffuse property for the shader

sh.diffuse=rgb(100,100,100)

sh.parallaxHeight = 0.035

-- Note: 0.035 is the standard value for parallax height. The value can be between 0
to 1. Where 0 implies no parallax and 1 implies maximum parallax.
-- The feel of depth increases by increasing this value.

-- Note: You can also change the shader material properties such as specular, ambient,
emissive, and shininess.

-- Assigning the parallax map shader to the 3D model.
member("3DMember").model("3DMtexture odel").shaderList = sh
```

Texture Transform [NEW]

Parallax mapping supports texture transform.

```
member(whichCastmember).shader(whichShader).textureTransform
member(whichCastmember).model(whichModel).shader.textureTransform
member(whichCastmember).model(whichModel).shaderList{[index]}.textureTransform
```

Cube Mapping

Information: This is a 3D enhancement.

Cube mapping is a method of using a six-sided cube as the shape of the map. The model is projected onto the six faces of a cube and stored as six square textures, or unfolded into six regions of a single texture. A cube map texture contains 6 2D images, each one representing a particular face of a cube.

Usage

```
member(whichCastmember).newTexture(newTextureName ,#cubemapTexture, cubeEdgeLength,
sourceRef1, sourceRef2, sourceRef3, sourceRef4, sourceRef5, sourceRef6)
```

Each of `sourceRef1` to `sourceRef6` must be either a cast member with bitmap texture or a Lingo image object.. These members are applied to six different faces of the cube map texture as follows:

- ✓ `sourceRef1` gets mapped to positive X-side
- ✓ `sourceRef2` gets mapped to negative X-side
- ✓ `sourceRef3` gets mapped to positive Y-side
- ✓ `sourceRef4` gets mapped to negative Y-side
- ✓ `sourceRef5` gets mapped to positive Z-side
- ✓ `sourceRef6` gets mapped to negative Z-side

The cube map images must always have the square dimensions so that the faces can form a cube. Hence, the width and height of each bitmap image in the cube map must be equal.

Note: Two types of texture co-ordinates generation modes are supported that generates the eye-space reflection vector or normal vector in the `(s,t,r)` texture coordinates.

A new property is added to the "standard" shader to allow setting texture co-ordinate generation mode for cube map texture. To perform this, after setting the created texture to a standard shader, use the property "`textureCoordGenMode`" or "`textureCoordGenModeList`". The values allowed are `#normal` and `#reflection`. By default, it comes with the eye-space reflection effect, `#reflection`.

The property "`textureCoordGenMode`" modifies the values for all the eight layers of textures. Using Lingo indexing command "`textureCoordGenModeList`", you can set the value for individual texture layer.

Normal mapping

Information: This is a 3D enhancement.

Normal mapping is a technique used for showing finer details like lighting of bumps and dents without using additional polygons. A normal map texture provides the direction of the normal (used for lighting calculations) at each pixel on the rendered image.

Now, Adobe Director supports using normal mapping. Normal maps are exposed in Lingo as a new type of shader, which can be created by passing **#normalMap** as the type of shader to the '**newShader**' handler of the 3D cast member.

Normal Map Usage

The following example shows the usage of normal maps in Adobe Director:

```
      -- Creating the normal map texture
      normalMap = mem.newTexture("normalMap", #fromCastMember,
member("normalMap_Image"))


      -- Creating the diffuse texture
      my_diffuse_texture = mem.newTexture("texture1", #fromCastMember,
member("texture_Image"))


      -- Creating the specular texture
      my_specular_texture = mem.newTexture("my_specular_texture",
#fromCastMember, member("specular_Image"))


      -- Creating a normal map shader
      sh = mem.newShader("myShader", #normalMap)


      -- Assign the normal map texture to the first layer
      sh.textureList[1] = normalMap


      -- Assign the diffuse texture to the second layer
      sh.textureList[2] = my_diffuse_texture


      -- Assign the specular texture to the third layer
      sh.textureList[3] = my_specular_texture


      -- Note: You can also use layers 4,5, and 6 for any other textures


      -- Applying diffuse property for the shader
      sh.diffuse=rgb(100,100,100)
```

```
-- Note: You can also change the shader material properties such as
specular, ambient, emissive, and shininess.


-- Assigning the normal map shader to the 3D model.
member("3DMember").model("3DModel").shaderList = sh
```

Texture Transform [NEW]

Normal mapping supports texture transform.

```
member(whichCastmember).shader(whichShader).textureTransform
member(whichCastmember).model(whichModel).shader.textureTransform
member(whichCastmember).model(whichModel).shaderList{[index]}.textureTransform
```

Camera enhancements

Information: This is a 3D enhancement.

A new property has been introduced in this release, **depthBuffer.clearAtRender**.

depthBuffer.clearAtRender, 3D property; indicates whether the depth buffer is cleared after each frame. Setting the value to **FALSE** means that the depth buffer is not cleared. The default value for this property is **TRUE**.

Usage

```
-- Lingo
member(whichCastmember).camera(whichCamera).depthBuffer.clearAtRender
sprite(whichSprite).camera{(index)}.depthBuffer.clearAtRender
```

Example

```
-- Lingo syntax
sprite(1).camera.depthBuffer.clearAtRender = 0
// JavaScript syntax
sprite(1).camera.getPropRef("depthBuffer").clearAtRender = 0;
```

The previously mentioned example prevents Adobe Director from erasing past content of the depth buffer.

Also, two new properties have been added:

- ✓ `Camera.overlay.scaleX`
- ✓ `Camera.overlay.scaleY`

Now, you can scale the overlay/backdrop textures individually on X and Y axis.

Render to Texture

Information: This is a 3D enhancement.

Render Textures are textures that are created and updated at runtime. In Adobe Director, to work with Render Textures, perform the following tasks:

- ✓ Create a Render Texture object. For creating a Render Texture object, invoke the `<3dCastMemRef>.newTexture()` method with `#typeIndicator` as `#renderTexture` and pass the dimensions of the Render Texture. For instance:

```
myRT = member("3D World").newTexture("MyRT1",#renderTexture, 512, 512)
```

This code will create a new texture object of type `#renderTexture`. This object can be used like any other texture object. For instance, you can attach it to a shader or create camera backdrops or overlays .

Note: You can also get the image of a Render Texture by using the `<renderTexObj>.image` syntax. This type of image access is not supported for other texture types.

- ✓ Once the Render Texture object is created, it can be used to render a 3D scene into the Render Texture. You need to invoke the `<sprite>.renderToTexture(<camera>, <renderTexObj>)` method by passing the camera object and the destination Render Texture object for which the scene needs to be rendered.

API Details

The following APIs have been introduced/enhanced:

- ✓ **updateRTImageOnRender**
- ✓ newTexture (This method has been enhanced)
- ✓ **renderToTexture**
- ✓ **<RenderTextureObj>.image**

updateRTImageOnRender (Getter/Setter)

When a `renderToTexture()` method is invoked:

1. The scene is rendered on to a texture object present on the VRAM of the graphics card. This texture can be directly applied on to any of the models for future rendering.
2. From VRAM, a copy of the texture is made in Lingo so that it can be accessed by `<rendertexture>.image` after the `renderToTexture()` call.

Note: This step is time consuming and will happen only if `updateRTImageOnRender` is set to true.

The second step is not necessary if you do not need to read the `<rendertexture>.image` after the call. In that case, you can set the `updateRTImageOnRender` to 'false' before calling the `renderToTexture()` method to avoid a time consuming copy operation from VRAM to Lingo image.

The default value of this property is true.

newTexture

This method creates a new texture within the referenced member's texture palette and returns a reference to the new texture. The only way cast member textures will work is by specifying the cast member in the `newTexture` constructor.

Usage

```
member(whichCastmember).newTexture(newTextureName {,#typeIndicator,
sourceObjectReference})
```

The following syntax should be followed when creating a render texture object:

Syntax

```
member(whichCastmember).newTexture(newTextureName {,#typeIndicator, rendTexWidth,
rendTexHeight})
```

Parameters

- **newTextureName** (Required). Specifies the name of the new texture. The name of the new texture must be unique in the referenced cast member's texture palette.
- **typeIndicator** (Optional). Specifies the type of the new texture. If omitted, the new texture is created with no specific type. Valid values include the following:
  - #fromCastMember (a cast member)
  - #fromImageObject (a Lingo image object)
  - #renderTexture (create a render texture object)
- **sourceObjectReference** (Optional). Specifies a reference to the source cast member or Lingo image object. If omitted, the new texture is created from no specific source. `sourceObjectReference` must refer to a cast member if `typeIndicator` is `#fromCastMember`, and it must refer to a Lingo image object if `typeIndicator` is `#fromImageObject`. This parameter is not valid when the `typeIndicator` is `#renderTexture`. In this case, it is replaced by a different set of parameters (rendTexWidth, rendTexHeight).
- **rendTexWidth** (Required if `typeIndicator` is `#renderTexture`). Specifies the width of the Render Texture to create. Currently only power of 2 texture sizes are supported. Any other value will be automatically converted to the nearest power of 2.
- **rendTexHeight** (Required if `typeIndicator` is `#renderTexture`). Specifies the height of the Render Texture to create.

Note: Currently, texture sizes of the power of 2 are supported. Any other value will be automatically converted to the nearest power of 2.

The first line of the following example creates a new texture called Grass 02 from cast member 5 of castlib 1. The second line creates a blank new texture called Blank.

Example

```
-- Lingo
member("3D World").newTexture("Grass 02",#fromCastMember,member(5,1))
member("3D World").newTexture("Blank")

-- To Create a render texture
myRT = member("3D World").newTexture("MyRT1",#renderTexture, 512, 512)

// Javascript
member("3D World").newTexture("Grass 02",symbol("fromCastMember"),member(5,1)) ;
member("3D World").newTexture("Blank") ;
```

renderToTexture

This method renders the 3D sprite into the Render Texture object as viewed from the provided camera object.

Usage

```
<3dSpriteRef>.renderToTexture(cameraObj, renderTextureObj)
```

Parameters

- ✓ **cameraObj** (Required). Specifies the camera object through which to render the scene into the `renderTextureObj`. The camera object should be part of the sprite/member on which this method is being called.
- ✓ **renderTextrureObj** (Required). Specifies the Render Texture object into which the resulting scene image needs to be rendered.

The following example creates a new Render texture object and renders the 3D scene into it using the current camera of the sprite.

Example

```
-- Lingo
sprite3D = sprite("3D World")
myRT = member("3D World").newTexture("MyRT1",#renderTexture, 512, 512)
sprite3D.renderToTexture(sprite3D.camera, myRT)
```

<RenderTextureObj>.image (Getter/Setter)

This method returns the contents of the Render Texture object as a Lingo image.

Usage

```
<RenderTextureObj>.image
```

The following example starts with creating a new Render Texture object and then renders a 3D scene into it using the current camera of the sprite. Then the rendered image is populated into a bitmap cast member named **bitmapRT.**

Example

```
-- Lingo
sprite3D = sprite("3D World")
myRT = member("3D World").newTexture("MyRT1",#renderTexture, 512, 512)
sprite3D.renderToTexture(sprite3D.camera, myRT)
member("bitmapRT").image = myRT.image
```

Limitations

- ✓ Texture dimensions can only be powers of 2. Any other value will be automatically set to the nearest power of 2.
- ✓ The only **renderFormat** supported for render textures is **#rgba8888**
- ✓ Saving the world with Render Texture objects is not supported.
- ✓ Antialiasing issue with Direct X.
- ✓ Only Direct-To-Stage is supported with this feature.
- ✓ At any given time, the same Render Texture cannot be used as both a rendering destination as well a source texture for models in the scene. The result is undefined and may be inconsistent across platforms. For instance, during a call to **renderToTexture()** to render a scene on to the Render Texture 'rt', models in the scene being rendered cannot have 'rt' as their texture.

Note: OpenGL support is provided in Windows and Mac 32-bit machines. Also, support is provided for non-power of two textures on Windows platform. The Render Texture dimensions can be of any size as long as it doesn't exceed the size of the sprite.

Bump mapping to normal mapping

Information: This is a 3D enhancement.

Bump mapping allows you to simulate bumps on the surface of an object. This effect is achieved by modifying the surface normals of the object.

Now, Adobe Director supports converting bump maps to normal maps.

bumpMapToNormalMap(<image>, bumpFactor)

This handler converts a gray scale bump map image to a normal map image and returns the normal map. Also, using this technique, you can create another normal map texture from the image returned from this API and you can apply the resultant texture to the normal map shader.

Parameters

- ✓ **image** (Required). A bump map image.

- ✓ **bumpFactor** (Required). An integer value specifying the bump value. The value can range from 0 to 1000.

The following example converts a bump map image of a member, 'bumpmap' to a normal map image with the bumpFactor, '1' and creates a normal map shader out of it.

Example

```
-- Lingo
normalMapImg = bumpMapToNormalMap(member("bumpmap").image, 1)
normalMapTex = member("3d").newTexture("NMTex", #fromImageObject,normalMapImg)
normalMapShader = member("3d").newShader("NMShader", #normalMap)

// JavaScript
normalMapImg = bumpMapToNormalMap(member("bumpmap").image, 1);
normalMapTex = member("3d").newTexture("NMTex", symbol("fromImageObject"),
normalMapImg);
normalMapShader = member("3d").newShader("NMShader", symbol("normalMap"));
```

Additive shading

Information: This is a 3D enhancement.

In this release, user can set `member("3D").shader("sh").multiTexturing = FALSE` to disable DirectX 9-based multi texturing (composing all texture layers in a single pass) for the given shader. This feature enables the user to make use of the old style additive shading and also the new multi-texturing in the same scene.

Note: This is an extension to `getRendererServices().multiTexturing` property, which earlier only allowed to disable multi-texturing on the whole scene, making it impossible to use single pass layered shading for other shaders in the same scene.

New/Modified APIs

- ➢ `getRendererServices().multiTexturing`
- ➢ `shader.multiTexturing` **[New Property]**

Note: If no value is specified for `shader.multiTexturing`, then the value assigned to `getRenderServices().multiTexturing` will be used. The default value of `getRenderServices().multiTexturing` is TRUE.

Alpha sorting

Information: This is a 3D enhancement.

Alpha sorting allows the use of transparent or semi-transparent polygons of a mesh to display in front of or behind one another according to their distance from the camera.

<shader>.alphaSort

This property can be set/read for a shader of type `#standard` or `#normalMap`. When the shader's 'transparent' property is set to `TRUE` (default), the polygons of the mesh are not alpha sorted for performance reasons. Hence, `alphaSort` property is FALSE by default. When the shader transparency is enabled, users can enable the inter-polygon alpha sorting by setting this property to `TRUE`.

Caution: Enabling alpha sorting by setting this property may some performance bottlenecks. Set this property only when it is absolutely required.

The following example shows how you can enable alpha sorting for the shader, 'tree' in the 3D cast member , '3d'

Usage

```
-- Lingo
member("3d").shader("tree").alphaSort = TRUE

// JavaScript
member("3d").shader("tree").alphaSort = true;
```

Enhancement of the visibility property

The `visibility` property allows you to get or set the visibility of the referenced model. This property determines how the model's geometry is drawn. It can have one of the following values:

- ✓ `#none.` Specifies that no polygons are drawn and the model is invisible.
- ✓ `#front`. Specifies that only those polygons facing the camera are drawn. This method is referred to as 'back-face culling' and it improves the rendering speed. This is the default setting for the property.
- ✓ `#back`. Specifies that only those polygons facing away from the camera are drawn. Use this setting when you want to draw the inside of a model, or for models which are not drawing correctly, possibly because they were imported from a file format that used a different handiness value when computing normals.
- ✓ `#both`. Specifies that both sides of all polygons are drawn. Use this setting when you want to see the plane regardless of the viewing direction, and for models that are not drawing correctly.
- ✓ `#front_onlydepth`. Same as `#front` but draws only to depth buffer and not to color buffer.
- ✓ `#back_onlydepth`. Same as `#back` but draws only to depth buffer and not to color buffer.
- ✓ `#both_onlydepth`. Same as `#both` but draws only to depth buffer and not to color buffer.

Usage

```
member(whichCastmember).model(whichModel).visibility
modelObjectReference.visibility
```

The following statement shows that the visibility property of the model Monster02 is set to `#none` to make the model invisible.

Example

```
-- Lingo syntax
put member("3D").model("Monster02").visibility
-- #none

// JavaScript syntax
trace(member("3D").getPropRef("model",1).visibility)
// symbol("none")
```

textureRepeat enhancements

The `textureRepeat` property controls the texture clamping behavior of the first texture layer of the shader. If `textureRepeatU` is set to true, the U component of the texture will be repeated. Similarly, if the `textureRepeatV` is set to true, the V component of the texture will be repeated.

Usage

```
member(whichCastmember).shader(whichShader).textureRepeatU
member(whichCastmember).model(whichModel).shader.textureRepeatU
member(whichCastmember).model(whichModel).shaderList{[index]}.textureRepeatU
```

The following statement sets the `textureRepeatU` property of the first shader used by the model named `gbCyl3` to `TRUE`.

Example

```
member("scene").model("gbCyl3").shader.textureRepeatU = TRUE
member("scene").model("gbCyl3").shader.textureRepeatV = FALSE
```

Custom blend function for particle systems

The `blendFunction` property, allows you to set custom values of blend functions. This function determines how blending will be performed between the particles in the system.

The `blendFunction` property can have the following values:

- ✓ **#multiply**. Multiplies the RGB values of the texture layer by the color being used for blending.
- ✓ **#add**. Adds the RGB values of the texture layer to the color being used for blending, and then clamps the texture to 255.
- ✓ **#replace**. Prevents the texture from being blended with the color set by the shader's diffuse property.
- ✓ **#blend**. Combines the colors of the texture layer with the color being used for blending in the ratio set by the **blendConstant** property. This is the default value.

Usage

```
member(whichCastmember).modelResource(whichModelResource).blendFunction
```

Note: This enhancement applies only to the particle systems.

Example

```
particleModelRes = pWorld.newModelResource ( "particle", #particle )
particleModelRes.blendFunction = #add
```

Importing DAE models

Adobe Director now provides support for importing DAE models. No

Note: Support is provided from Collada version 1.4.1 and higher.

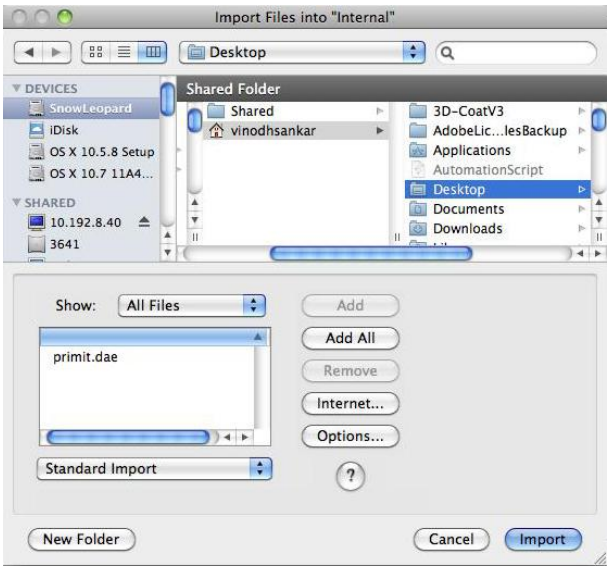
Not all features of DAE are supported. The currently supported features are:

- ✓ Support for meshes and models
- ✓ Support for Cameras
- ✓ Support for lights
- ✓ Support for textures
- ✓ Support for bones
- ✓ Support for animation

Note: From the **<effect>** tag of the Collada file, only **<profile_COMMON>** tag will be imported as a shader.

To import a DAE model, perform the following tasks:

- ✓ Click File > Import.
- ✓ Browse and select the .dae file.



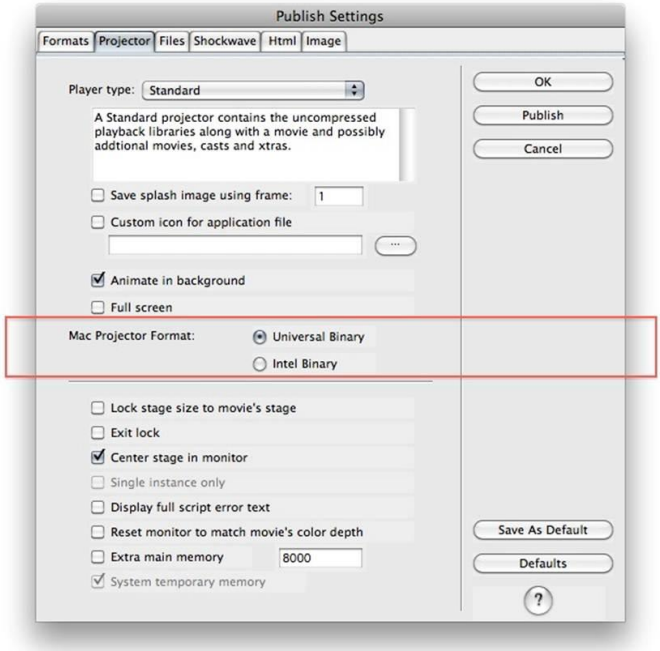
- ✓ Click Import.

Intel binary support

Adobe Director, now supports producing a Mac projector file in Intel binary format apart from the earlier supported universal binary format.

To publish Mac projector file in Intel binary, perform the following tasks:

- ✓ Click File > Publish Settings
- ✓ Click Projector tab
- ✓ For Mac Projector Format, select Intel Binary



✓ Click Publish